ARMY RESEARCH LABORATORY

Animation Techniques in BRL-CAD

Lee A. Butler
Christine Murdza

ARL-TR-313

December 1993

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1993	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Animation Techniques in BRL-CAD	5. FUNDING NUMBERS PR: 1L162618AH80
6. AUTHOR(S) Lee A. Butler and Christine Murdza	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: AMSRL-SL-BV Aberdeen Proving Ground, MD 21005-5066	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: AMSRL-OP-CI-B (Tech Lib) Aberdeen Proving Ground, MD 21005-5066	10. SPONSORING/MONITORING AGENCY REPORT NUMBER ARL-TR-313
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Complex model systems can be difficult to depict using static images. Some phenomena are not apparent when viewed at a single instance in time. Many such things become much clearer, easier to detect and understand when they can be presented as moving, changing, evolving entities. For this reason, a set of tools have been created to help users generate motion picture sequences of systems modeled with BRL-CAD.

Raytracing is a fundamental analysis and rendering technique within BRL-CAD. Most users of BRL-CAD are acquainted with the use of the program "rt" for creating still images. A relative few understand how to utilize its capabilities for generating animation sequences or "movies." This paper introduces some of the tools and techniques available for creating animation sequences with "rt." A basic knowledge of BRL-CAD, the geometry editor "mged" and image creation using "rt" is assumed.

14. SUBJECT TERMS BRL-CAD, animation, video tape recording, ray tracing	15. NUMBER OF PAGES 35
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

INTENTIONALLY LEFT BLANK.

Table of Contents

INTENTIONALLY LEFT BLANK.

1. Introduction

Moving pictures are created by presenting the viewer with a sequence of still images in quick succession. Objects which occur in an orderly succession of slightly different locations within a sequence of images appear to be in motion to the viewer. Such an object is said to be ''animated.'' Preparing a moving picture of animated objects requires a large number of still images (often called ''frames'').

Still images of computer models are relatively easy to create within BRL-CAD.[1] The program rt uses the technique of ray-tracing to create images of geometric models.[2] A moving picture or ''animation'' of the model can be created by using rt to create a series of still images, each of which forms a frame of the final moving picture. The difficulty arises in specifying each frame for rt to create. Some tools have been created to help make this process easier.

1.1. Preliminaries

Before there can be motion there must first be form. This means a description of geometry to be animated must exist. To keep things simple, we will use the ''moss'' database distributed with BRL-CAD. Assuming that the distribution is stored in the directory **/cadsrc** the following command will create a local copy of the database for use with the examples presented here:

```
% asc2g < /cadsrc/db/moss.asc > moss.g
```

Examples such as the one above will be presented with the portion typed by the user shown in **bold** typeface.

2. Simple Camera Motion

2.1. Key Frames

The first step in creating an animation sequence is the definition of ''key-frames.'' These are descriptions of what the scene should look like at ''key'' moments in the animation sequence. Some key-frames are readily recognized. The frame in which the camera or an object starts or stops moving is important. Likewise, the moment when an object changes direction or velocity is also important. It is good to define one or two key-frames before and after each of these points of change.

Generating key-frames with mged is easy. The user displays the wireframe of the desired geometry and manipulates the display until the desired view is achieved. To save the key-frame information in a file, the mged keyboard command ''saveview'' is used. This command creates a shell script with the proper invocation of the rt program to render the current scene. The argument to the ''saveview'' command is the filename for the shell script. Note that the keyboard command ''saveview'' is distinct and different from the menu option visible in the button menu on the geometry display.

To keep things orderly and make later processing easier, it is recommended that the user specify key-frame filenames which have a common prefix followed by a number indicating the time in the animation sequence at which the key-frame occurs.

2.2. Generating Key Frames with MGED

For our first example, we will create a simple animation with the ''moss.g'' model. The objective is to begin the animation sequence with a view of the geometry from the front corner of the platform. As time goes by, the viewer's location (the ''eye point'' or ''eye_pt'') moves upward in an arc over the geometry until the viewer is looking directly down on the center of the platform. To give a natural feeling of flight, the eye point should accelerate and decelerate at the beginning and end of its travel respectively. This is achieved by adjusting the number of degrees of elevation the eye will raise in each second of the animation. In the first (and last) three quarters of a second of the animation, the eye raises only 2 degrees of elevation above the plane. In the following second 8 degrees of arc are covered. At the 4 second mark the eye is looking down at a 45 degree angle.

[1] See **[Deitz89]** for an overview of BRL-CAD.

[2] See **[Muuss88a]** for a discussion of modeling and raytracing within BRL-CAD, and **[Muuss88b]** for a discussion of the rt lighting model.

```
% mged moss.g
BRL-CAD Release 4.1   Graphics Editor (MGED)
    Tue Oct 20 14:19:59 EDT 1992, Compilation 5
    stay@vail:/n/wolf/m/dist4.1/mged

attach (nu|tek|tek4109|ps|plot|sgi|X)[nu]? sgi
ATTACHING sgi (SGI 4d)
Gary Moss's "World on a Platter" (units=mm)
mged> e all.g
408 vectors in 1 sec
mged> center 20 0 0
mged> size 200
mged> ae 45 0
mged> saveview moss_0
mged> ae 45 2
mged> saveview moss_0.75
mged> ae 45 10
mged> saveview moss_1.75
mged> ae 45 45
mged> saveview moss_4
mged> ae 45 80
mged> saveview moss_6.25
mged> ae 45 88
mged> saveview moss_7.25
mged> ae 45 90
mged> saveview moss_8
mged> q
%
```

At this stage there are seven key-frame files in the current directory with the names:

```
moss_0      moss_1.75   moss_6.25   moss_8
moss_0.75   moss_4      moss_7.25
```

A look at the contents of one of the key-frame files shows that the scene is stored as four separate elements. These elements are the geometry being displayed, the size of the viewing cube or ''viewsize'', the location in 3 dimensional space of the camera or ''eye_pt'', and the ''orientation'' of the geometry within space. All animations which involve the observer moving about a static object can be generated from these parameters.

These key-frames do not represent all of the images necessary to produce the animation. They only specify the significant moments in the animation. It is necessary to generate the ''in-between'' frames as well to turn the key-frames into a smooth animation. The values for the ''viewsize,'' ''eye_pt'' and ''orientation'' attributes of these ''in-between'' frames are generated by interpolating the values which describe the key-frames.

2.3. Key Frame Interpolation

BRL-CAD has a utility for performing interpolation called ''tabinterp.'' It operates on files containing columns of numbers. The left-most column is always used to hold the time in the animation at which the data in the other columns occurs. A column is referred to as a ''channel.'' Lines beginning with a ''#'' character are considered to be comments, and are ignored by tabinterp. Table A shows an example input file for tabinterp.

Table A			
#Time	X	Y	Z
0	100	0	0
1	74.24	51.98	42.23
2	-91.85	91.85	75

This table has four channels (columns). The leftmost channel is always the ''time channel''. In this instance the ''time channel'' has the values 0, 1, and 2. The time channel has no inherent unit associated with it. These values could represent microseconds, seconds, minutes, etc. depending upon the motion being specified. The second column contains the first actual data channel in this file. In this instance it is an X coordinate. There are three data channels in this file. The contents of the file are said to form an interpolation table.

To create the ''in-between'' frames of our animation, we need to extract the values which are arguments to the ''viewsize'', ''eye_pt'', and ''orientation'' commands to rt stored in the key-frame files. These values must be tabulated for input to tabinterp.

While it would be possible to concatenate the key-frame files and edit the result by hand to create the table, this would be tedious for all but the simplest animations. Instead it is recommended that a shell script such as the ''key-chans'' script shown below be employed to do the work.

```
#!/bin/sh
if [ "$#" != "1" ] ; then
      echo "Usage: $0 basename"
      exit
fi
awk '/^viewsize/ { print FILENAME " " $2}' $1* |\
   sed -e 's/;//' -e "s/^$1//" | sort -n > chans.vsize

awk '/^eye_pt/ { print FILENAME " " $2 " " $3 " " $4}' $1* |\
   sed -e 's/;//' -e "s/^$1//" | sort -n > chans.eyept

awk '/^orient/ { print FILENAME " " $2 " " $3 " " $4 " " $5}' $1* |\
   sed -e 's/;//' -e "s/^$1//" | sort -n > chans.orient
```

This shell script creates the three files ''chans.vsize'', ''chans.eyept'', and ''chans.orient'' from key-frame files which share the same basename (such as ''moss_''). The shared basename is specified on the command line.

```
% key-chans moss_
```

The files ''chans.vsize,'' ''chans.eyept,'' ''chans.orient'' now contain the values needed for interpolation. The file ''chans.vsize'' contains the argument to the ''viewsize'' directive in the key-frame files. Likewise, ''chans.eyept'' contains the X, Y, and Z arguments to the ''eye_pt'' directive and ''chans.orient'' contains the quaternion[3] argument to the ''orientation'' directive.

chans.vsize	
0	2.000000000000000e+02
0.75	2.000000000000000e+02
1.75	2.000000000000000e+02
4	2.000000000000000e+02
6.25	2.000000000000000e+02
7.25	2.000000000000000e+02
8	2.000000000000000e+02

[3] See **[Shoemake85]** for a description of quaternions and their use.

chans.eyept			
0	9.071067811865476e+01	7.071067811865474e+01	0.000000000000000e+00
0.75	9.066760308408352e+01	7.066760308408351e+01	3.489949670250149e+00
1.75	8.963642403200188e+01	6.963642403200188e+01	1.736481776669301e+01
4	6.999999999999997e+01	4.999999999999999e+01	7.071067811865471e+01
6.25	3.227878039689727e+01	1.227878039689727e+01	9.848077530122080e+01
7.25	2.246776707783357e+01	2.467767077833573e+00	9.993908270190957e+01
8	2.000000000000000e+01	0.000000000000000e+00	9.999999999999999e+01

chans.orient				
0	2.705980500730985e-01	6.532814824381883e-01	6.532814824381883e-01	2.705980500730985e-01
0.75	2.658342495283891e-01	6.417806505547106e-01	6.645833184536355e-01	2.752794238304135e-01
1.75	2.459841687565966e-01	5.938583163412476e-01	7.077327819916303e-01	2.931525168369743e-01
4	1.464466094067262e-01	3.535533905932737e-01	8.535533905932737e-01	3.535533905932737e-01
6.25	3.335305878500257e-02	8.052140686538031e-02	9.203638919632243e-01	3.812272063696535e-01
7.25	6.678746798450165e-03	1.612392110047428e-02	9.237388211835743e-01	3.826251478247717e-01
8	0.000000000000000e+00	0.000000000000000e+00	9.238795325112867e-01	3.826834323650898e-01

Now the key-frame data can be interpolated. The tabinterp[4] program reads commands from standard input until end-of-file is found. Commands specify files from which interpolation tables should be read, which channels in the tables should be used, what type of interpolation should be applied, and the range of time over which values are needed. When end of file is reached, tabinterp performs any interpolations requested and writes out the requested results.

```
% tabinterp << EOF > chans.all
file chans.vsize 0;
file chans.eyept 1 2 3;
file chans.orient 4 5 6 7;
times 0 8 3;
interp spline 0 1 2 3 4 5 6 7;
EOF
cmd: file chans.vsize 0
chan 0:  File 'chans.vsize', Column 1
cmd: file chans.eyept 1 2 3
chan 1:  File 'chans.eyept', Column 1
chan 2:  File 'chans.eyept', Column 2
chan 3:  File 'chans.eyept', Column 3
cmd: file chans.orient 4 5 6 7
chan 4:  File 'chans.orient', Column 1
chan 5:  File 'chans.orient', Column 2
chan 6:  File 'chans.orient', Column 3
chan 7:  File 'chans.orient', Column 4
cmd: times 0 8 3
cmd: interp spline 0 1 2 3 4 5 6 7
performing interpolations
writing output
%
```

In this instance, the data from the ''viewsize'' interpolation table is read into channel 0 of tabinterp. The eye point data is acquired from another file to fill channels 1, 2, and 3. The orientation table data are read into channels 4, 5, 6, and 7. The user command ''times 0 8 3'' indicates that interpolation is to be performed for the time sequence starting at time 0, through time 8 with 3 frames per time step. If each time step represents a second, this is

[4] A description of tabinterp can be found in Appendix A.

not enough frames per second for a finished product such as a videotape, but it will be sufficient to create a preview of the sequence. It would not be wise to expend the CPU time required to compute all of the frames before we are certain that we have the sequence correct.

chans.all								
0	200	90.7107	70.7107	0	0.270598	0.653281	0.653281	0.270598
0.333333	200	90.6992	70.6992	1.11757	0.26908	0.649617	0.656908	0.2721
0.666667	200	90.6764	70.6764	2.87542	0.266677	0.643815	0.662597	0.274457
1	200	90.6193	70.6193	5.86093	0.262565	0.633889	0.672216	0.278441
1.33333	200	90.4026	70.4026	10.2013	0.256455	0.619137	0.685929	0.284121
1.66667	200	89.8485	69.8485	15.7869	0.248331	0.599525	0.703018	0.291199
2	200	88.7905	68.7905	22.4928	0.238198	0.575062	0.722757	0.299376
2.33333	200	87.1626	67.1626	30.0787	0.226209	0.546117	0.744384	0.308334
2.66667	200	84.9511	64.9511	38.245	0.212592	0.513243	0.76712	0.317752
3	200	82.1425	62.1425	46.6914	0.197578	0.476996	0.790185	0.327305
3.33333	200	78.7234	58.7234	55.118	0.181396	0.437929	0.812799	0.336672
3.66667	200	74.6804	54.6804	63.2244	0.164276	0.396596	0.834181	0.345529
4	200	70	50	70.7107	0.146447	0.353553	0.853553	0.353553
4.33333	200	64.7064	44.7064	77.3298	0.128166	0.309419	0.870291	0.360486
4.66667	200	58.9743	38.9743	83.0475	0.109797	0.265073	0.884398	0.36633
5	200	53.0158	33.0158	87.8828	0.091731	0.221458	0.896032	0.371149
5.33333	200	47.0433	27.0433	91.8548	0.0743588	0.179518	0.905354	0.37501
5.66667	200	41.2689	21.2689	94.9823	0.0580711	0.140196	0.912522	0.377979
6	200	35.9048	15.9048	97.2844	0.0432589	0.104436	0.917696	0.380122
6.33333	200	31.1631	11.1631	98.7807	0.0303124	0.0731806	0.921036	0.381506
6.66667	200	27.2134	7.21344	99.5643	0.0195628	0.0472287	0.922821	0.382245
7	200	24.1443	4.1443	99.8707	0.011226	0.0271019	0.923556	0.382549
7.33333	200	22.0332	2.03323	99.9515	0.00550101	0.0132806	0.923773	0.382639
7.66667	200	20.7902	0.790243	99.9837	0.00213547	0.00515548	0.923852	0.382672
8	200	20	0	100	0	0	0.92388	0.382683

The last command to tabinterp selects a spline interpolation for channels 0 through 7. When tabinterp runs out of commands to process it performs the interpolation. The table ''chans.all'' shows the result of the interpolation.

Each column in chans.all represents one channel from the ''tabinterp'' session. The leftmost column is always the time channel. The column next to that is data channel 0 from the tabinterp session. An examination of the time channel on the left will reveal that the sequence runs from time 0 through time 8 and that there are 3 lines (frames) for every integer time step.

This example used only spline interpolation. There are other interpolation techniques available including step, linear, and circular spline (cspline). With step interpolation, a channel value is simply copied to intermediate time points until a new value is encountered. The circular spline technique makes certain that the starting and stopping conditions of the time sequence are identical. The second derivative of the curve at the endpoints are made to be the same. This is useful when a seamless loop of values is desired.

2.4. Building the RT Animation Script

The program ''tabsub'' uses the output from a run of tabinterp along with a template file to write an animation script for rt. There are many types of ''macros'' which tabsub recognizes in the template. An understanding of two of these is necessary for the current example.[5] The character ''@'' followed by an integer (such as @2) is replaced by data from the interpolation channel of that number. Note that ''@0'' refers to values from the first data channel from the interpolation. It does **not** refer to the time channel. The macro ''@(line)'' is replaced with the line number of the file from which the data was read. The @(line) macro serves primarily to indicate the animation

[5] A description of tabsub and the macros it recognizes can be found in Appendix B.

frame number. A template file (''moss.proto'' for this example) should be created:

```
% cat moss.proto
viewsize @0;
eye_pt @1 @2 @3;
orientation @4 @5 @6 @7;
start @(line);
end;
```

This file can be used in conjunction with tabsub to create an animation script for rt.

```
% tabsub moss.proto chans.all > moss.rtanim
```

The resultant file ''moss.rtanim'' is rather long, so for illustration purposes only the commands for the first two frames are shown here.

```
% head -12 moss.rtanim
viewsize 200;
eye_pt 90.7107 70.7107 0;
orientation 0.270598 0.653281 0.653281 0.270598;
start 0;
end;

viewsize 200;
eye_pt 90.6992 70.6992 1.11757;
orientation 0.26908 0.649617 0.656908 0.2721;
start 1;
end;
```

2.5. Previewing Animations with MGED

Under Version 4.2 of BRL-CAD and beyond, the user has the ability to preview animations using mged. This is a useful check to run before expending the CPU time to compute the images with rt.

```
% mged moss.g
BRL-CAD Release 4.2   Graphics Editor (MGED)
    Wed Nov 11 00:36:43 EST 1992, Compilation 770
    mike@wolf.brl.mil:/m/cad/.mged.5d

attach (nu|tek|tek4109|ps|plot|sgi)[nu]? sgi
ATTACHING nu (Null Display)
Gary Moss's ''World on a Platter'' (units=mm)
mged> preview moss.rtanim
tree
eyepoint at (0,0,1) viewspace
db_lookup:  could not find 'EYE_PATH*'
mged> e all.g
```

Don't let the ''db_lookup'' message frighten you. This is just mged telling you that there wasn't an EYE_PATH overlay prior to your first ''preview'' command. This is the name of the ''pseudo-object'' that mged creates for storing the overlay. This object is not written into the geometry database. The preview command paints a small ''L'' shape at the eye point for each frame and connects all the corners together. The location of the ''L'' indicates the location of the eye for the frame. The orientation of the ''L'' is in the plane perpendicular to the viewing direction for the frame.

Look carefully at the path the eye will take during the animation. In this instance we want the path to form an arc from near one of the corners of the platform to a point directly over the top of the platform. Does it look like what was intended? Does the arc seem to pass through any of the objects? On a machine which supports fast polygon rendering, such as the Silicon Graphics Iris, displaying the geometry with the ''ev'' command instead of the ''e'' command will make finding eye/geometry collisions much easier.

2.6. Creating Postage Stamp Animations

Once you are convinced that the eye path is correct you are ready to produce a test animation. The file ''moss.rtanim'' can be fed directly to rt either from the command line or in a shell script. The invocation of moss.rt shown below will generate images that are 200 pixels square. This is a good size to use for an initial rendering. It will let us preview our animation before committing the resources to generate the complete animation.

```
% cat moss.rt
#!/bin/sh
rt -M $* -o moss.pix moss.g 'all.g' 2>> moss.log < moss.rtanim

% moss.rt -s 200
```

Once the frames have been computed, they can be turned into a ''postage stamp animation''. The ''pixtile'' program takes many small images and creates a bigger image from a mosaic of the small images (See Plate 1 for the image created by the pixtile command below). This image can then be displayed on a framebuffer, and the animation sequence played using the ''fbanim'' command.

```
% mv moss.pix moss.pix.0
% fbserv -S 1024 0 /dev/sgip &
% setenv FB_FILE :0
% pixtile -s 200 -S 1024 moss.pix | pix-fb -h
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
% fbanim -S1024 -s200 -p5 200 25 3
```

The fbanim program will show the animation sequence 5 times (the ''-p5'' option specifies number of passes through the sequence). When fbanim finishes it will leave the framebuffer zoomed in on the last image. To unzoom the framebuffer run the ''fbzoom'' program and give it the ''r'' command. This tells fbzoom to reset the framebuffer to normal. Typing ''q'' will cause fbzoom to exit.

3. Object Motion

Animations involving just camera motion through a static scene or around a static object are adequate for many different applications. Using only camera motion, the viewer can see what it would be like to walk through a building which exists only as a computer model. The view from the driver's seat of a new vehicle design can be created. Even a simulation of an object flying past the viewer can be created (by flying the viewer past the object instead).

For a variety of applications, the true value of animation is realized only when the geometric objects themselves have motion. Perhaps a vehicle drives past a familiar stationary landmark and the eye follows the vehicle away after it enters the scene. As the vehicle drives over a bump in the road the suspension system is seen to flex and energy is transferred to the frame. Things which cannot ordinarily be seen can be made more visible by watching them change over time. As our vehicle crosses a bridge, its weight causes it to flex and vibrate even after the vehicle is gone. These effects could be made visible by amplifying them until they are readily observable.

3.1. Matrix Manipulations

The animation of objects is accomplished by specifying matrix transformations to be applied to database elements before each image is calculated. This allows any solid or combination in the model to have its definition independently rotated, moved, or resized as the animation proceeds.

In BRL-CAD matrices are stored in a traditional mathematics form. This is different from (the transpose of) the form used in most texts on computer graphics. In BRL-CAD matrices are stored as follows:

$$\begin{bmatrix} r1 & r2 & r3 & dx \\ r4 & r5 & r6 & dy \\ r7 & r8 & r9 & dz \\ 0 & 0 & 0 & 1/s \end{bmatrix}$$

As a result, points and vectors in 3 dimensional space are represented as 4-tuple *column* vectors. Points and vectors are therefore properly transformed by the matrix equation:

$$\begin{bmatrix} X'/\omega' \\ Y'/\omega' \\ Z'/\omega' \\ 1 \end{bmatrix} = \begin{bmatrix} r1 & r2 & r3 & dx \\ r4 & r5 & r6 & dy \\ r7 & r8 & r9 & dz \\ 0 & 0 & 0 & 1/s \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \\ \omega \end{bmatrix}$$

Individuals who are unfamiliar with matrix transformations in general and homogeneous coordinate systems in specific are urged to study **[Rogers90]** or **[Newman79]** or **[Foley92]**.

The matrix operations in a BRL-CAD database can be thought of as living in the arcs of the directed acyclic graph. In slightly simpler terms, the matrix lives between an object (either primitive solid or combination record) and the parent combination record in the model tree. The model coordinate system is on the "left" end of a "stack" of matrix multiplications which is built up as the graph is traversed. When traversing the graph from the root to the leaves, matrices encountered on the arcs are applied to the "right" of the matrix equation. For example, the graph formed by "all.g" from our geometry file "moss.g" can be thought of as the following table:[6]

[MatrixA]	all.g	[MatrixPr]	platform.r	[MatrixPs]	platform.s
		[MatrixBr]	box.r	[MatrixBs]	box.s
		[MatrixEr]	ellipse.r	[MatrixEs]	ellipse.s
		[MatrixCr]	cone.r	[MatrixCs]	cone.s
		[MatrixTr]	tor.r	[MatrixTs]	tor
		[MatrixLr]	light.r	[MatrixLs]	LIGHT

The program rt would transform the origin (and other parameters) of "tor" with the following matrix equation:

$$tor.location = \begin{bmatrix} MatrixA \end{bmatrix} \times \begin{bmatrix} MatrixTr \end{bmatrix} \times \begin{bmatrix} MatrixTs \end{bmatrix} \times \begin{bmatrix} tor.X \\ tor.Y \\ tor.Z \end{bmatrix}$$

3.2. The RT Matrix Operations for Animation

Each of the matrices in the database can be altered individually during the animation. It is also possible to replace the "stack" matrix which has been accumulated. These operations are achieved with the "anim" command in the input script to rt. The command has the form:

anim *Path* matrix *Operation [Matrix]*;

where *Path* specifies the arc where the operation takes place. Either a specific use of the matrix within the model, or all uses of an arc within the model can be specified.

The *Operation* portion of the anim command specifies the matrix operation to be performed. The set of valid operations is listed below.

Anim Matrix Operations	
rstack	replace the entire stack up to this point
rarc	replace the matrix on the specified arc
rboth	replace the entire stack and the arc matrix
rmul	arc_matrix = arc_matrix * matrix
lmul	arc_matrix = matrix * arc_matrix

[6] Purists will note that MatrixA is not directly stored in the database. It exists as a conceptual aid to editing models and creating animations. In reality MatrixA will be combined with each of the matrices MatrixPr, MatrixBr, MatrixEr, MatrixCr, MatrixTr, MatrixLr from the table above.

For example, the following command always replaces the matrix on the arc between "arm" and "hand" with a new matrix:

```
anim arm/hand matrix rarc

1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1;
```

Whereas in the next example the matrix will be replaced only when "arm/hand" occurs as a direct child of "body/left" in the database tree. This would permit the left and right hands to be modeled as different instances of a single hand prototype, and still allow the left hand to be manipulated without affecting the right hand.

```
anim body/left/arm/hand matrix rarc

1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1;
```

Finally, a command of the form:

```
anim hand matrix rarc

1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1;
```

operates on any arc which ends in a node called "hand" regardless of where it occurs in the model hierarchy. Note that element "hand" in these examples above is not a leaf node (primitive solid) in the model graph.[7]

All object parameters in the database are stored using millimeters as the unit of measure. As a result, all matrix operations are carried out in units of millimeters. It is important to remember this when preparing matrices for use with the rt anim command. An example rt session will serve to illustrate the proper use of the anim command.

The following shell script "trans.sh" runs rt to create two separate images.[8] The first is saved in the file "trans.pix.1" and is approximately the view selected for the key-frame "moss_8" in Section 2.

```
#!/bin/sh
rt -M $* -o trans.pix moss.g 'all.g' 2>> trans.log <<EOF
viewsize 200;
eye_pt 20.0 0.0 100;
orientation 0.0 0.0 0.924 0.383;
start 1;
clean;
end;

start 2;
clean;
anim all.g/tor.r matrix rarc
     1 0 0 0
     0 1 0 80
     0 0 1 0
     0 0 0 1;
end
EOF
```

[7] With version 4.2 of BRL-CAD and beyond the user will be able to manipulate the matrices on the arcs between primitive solids and their parent combinations.

[8] The "pix-fb" utility can be used to display these images on the framebuffer.

The second image is saved in "trans.pix.2" and is the same except that the matrix on the arc between "all.g" and "tor.r" is replaced with a new matrix. This matrix has the effect of translating the torus 80 millimeters along the Y axis of the model coordinate system. Plate 2 shows trans.pix.1 and Plate 3 shows trans.pix.2.

3.3. Preparing an Animation with Motion

It is time to re-visit the animation sequence we developed in Section 2. We are going to add animation of the objects in the scene to the existing eye-point movement already created. The ellipsoid will be given a constant velocity along a vector which will take it through the center of the torus.

To make the ellipsoid pass through the center of the torus we must determine the vector from the center vertex of "ellipse.s" to the center vertex of "tor"

```
% mged moss.g
BRL-CAD Release 4.1   Graphics Editor (MGED)
  Tue Oct 20 14:19:59 EDT 1992, Compilation 5
  stay@vail:/n/wolf/m/dist4.1/mged

attach (nu|tek|tek4109|ps|plot|sgi|X)[nu]? sgi
ATTACHING sgi (SGI 4d)
Gary Moss's "World on a Platter" (units=mm)
mged> l all.g
all.g:  all.g (len 6) --
 u platform.r
 u box.r [-23.6989,13.41,8.02399]
 u cone.r [22.0492,12.2349,2.11125e-07]
 u ellipse.r [14.6793,-41.6077,38.7988]
 u tor.r
 u light.r
mged> l ellipse.s
ellipse.s:  ellipsoid (ELL)
     V (16.1309, 46.6556, -3.72252)
     A (14.8761, 0, 0) mag=14.8761
     B (0, 8.98026, -8.98026) mag=12.7
     C (0, 8.98026, 8.98026) mag=12.7
     A direction cosines=(0.0, 90, 90)
     A rotation angle=0, fallback angle=0
     B direction cosines=(90.0, 45, 135)
     B rotation angle=90, fallback angle=-45
     C direction cosines=(90.0, 45, 45)
     C rotation angle=90, fallback angle=45

mged> l tor
tor:  torus (TOR)
     V (4.91624, -32.8022, 31.7118), r1=25.4 (A), r2=5.08 (H)
     N=(0, 1, 0)
     A=(0, 0, 1)
     B=(1, 0, 0)
     vector to inner edge = (0, 0, 20.32)
     vector to outer edge = (0, 0, 30.48)
mged> q
```

Doing a little vector math we find the vertex of the "ellipse.s" as it is found in "all.g" is at:

$$\begin{bmatrix} 16.1309 \\ 46.6556 \\ -3.7225 \end{bmatrix} + \begin{bmatrix} 14.6793 \\ -41.6077 \\ 38.7988 \end{bmatrix} = \begin{bmatrix} 30.8102 \\ 5.0479 \\ 35.0763 \end{bmatrix}$$

Now we subtract this from the origin of "tor" to get a vector that will translate "ellipse.s" to the center of the torus. This vector is scaled by a factor to 2 to get a ''net displacement'' vector for the ellipsoid.

$$\left(\begin{bmatrix} 4.9162 \\ -32.8022 \\ 31.7118 \end{bmatrix} - \begin{bmatrix} 30.8102 \\ 5.0479 \\ 35.0762 \end{bmatrix}\right) * 2 = \begin{bmatrix} -51.7879 \\ -75.7002 \\ -6.7289 \end{bmatrix}$$

We can now create a new interpolation table with motion values to be interpolated. The ellipse will start moving half a second after the sequence starts. It will reach its destination half a second before the end of the sequence. Assuming that the time channel is being specified in units of seconds, the result is the table chans.ellanim.

chans.ellanim			
0.5	0	0	0
7.25	-51.78792	-75.7002	-6.72896

The interpolation is done much as it was in Section 2. The data from ''chans.ellanim'' is read into interpolation channels 8, 9, and 10 within tabinterp. The use of linear interpolation ensures that the ellipse will move at a constant rate to its destination.

```
% tabinterp << EOF > chans.all
file chans.vsize 0;
file chans.eyept 1 2 3;
file chans.orient 4 5 6 7;
file chans.ellanim 8 9 10;
times 0 8 3;
interp spline 0 1 2 3 4 5 6 7;
interp linear 8 9 10;
EOF
cmd: file chans.vsize 0
chan 0:  File 'chans.vsize', Column 1
cmd: file chans.eyept 1 2 3
chan 1:  File 'chans.eyept', Column 1
chan 2:  File 'chans.eyept', Column 2
chan 3:  File 'chans.eyept', Column 3
cmd: file chans.orient 4 5 6 7
chan 4:  File 'chans.orient', Column 1
chan 5:  File 'chans.orient', Column 2
chan 6:  File 'chans.orient', Column 3
chan 7:  File 'chans.orient', Column 4
cmd: file chans.ellanim 8 9 10
chan 8:  File 'chans.ellanim', Column 1
chan 9:  File 'chans.ellanim', Column 2
chan 10:  File 'chans.ellanim', Column 3
cmd: times 0 8 3
cmd: interp spline 0 1 2 3 4 5 6 7
cmd: interp linear 8 9 10
performing interpolations
writing output
%
```

An appropriate template such as the one below must be created for use with tabsub. Note the use of the ''clean'' command to rt at the beginning of each frame in the template. This is required after the ''start'' for each frame in rt animation scripts which use the ''anim'' command. This command tells rt (librt actually) to forget any accumulated animation matrices, thereby restoring the geometry to the form it has in the database.

```
% cat ell.proto
viewsize @0;
eye_pt @1 @2 @3;
orientation @4 @5 @6 @7;
start @(line);
clean;
anim all.g/ellipse.r matrix rmul
        1 0 0 @8
        0 1 0 @9
        0 0 1 @10
        0 0 0 1;
end;

% tabsub ell.proto chans.all > ell.rtanim
```

We can preview the path that the ellipse will take by creating a plot file which can be used as an overlay in mged.[9]

```
% awk '{print $2+30.8102 " " $3+5.0479 " " $4+35.07628}' chans.ellanim | \
xyz-pl > ell.pl

% mged moss.g
BRL-CAD Release 4.2   Graphics Editor (MGED)
   Wed Nov 11 00:36:43 EST 1992, Compilation 770
   mike@wolf.brl.mil:/m/cad/.mged.5d

attach (nu|tek|tek4109|ps|plot|sgi|X)[nu]? sgi
ATTACHING sgi (SGI 4d)
Gary Moss's "World on a Platter" (units=mm)
mged> e all.g
408 vectors in 0.459896 sec
mged> overlay ell.pl
db_lookup:  could not find '_PLOT_OVER*'
mged>
```

The ''overlay'' command creates pseudo entries of the form '_PLOT_OVERLAY_' in the version of the database in memory. These are used to store the vectors of the overlay. They are never actually objects in the geometric database on disk. The next time the overlay command is given, the ''_PLOT_OVER*'' objects are removed and re-created. As a result, the message:

```
db_lookup:  could not find '_PLOT_OVER*'
```

is not a cause for concern.

Once again, a postage stamp animation can be created to view the positioning and motion of the camera and objects. Plate 4 shows the output of the pixtile command below.

```
% rt -M -s200 -o ell.pix moss.g 'all.g' >& ell.log < ell.rtanim
% mv ell.pix ell.pix.0
% pixtile -s 200 -S1024 ell.pix | pix-fb -h
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
% fbanim -h -p 5 200 25 3
```

[9] In version 4.2 of BRL-CAD and beyond the animation *sequence* can also be viewed using the mged ''preview'' command.

4. Making Frames for Video Tape

The use of ''postage-stamp'' animations with ''fbanim'' can provide useful visualization capabilities. Unfortunately, if there are enough frames to present a smooth sense of motion, the individual images are so small that details are lost. If the images are made larger, the motion is no longer smooth.

Videotape offers the ability to maintain a moderate image size (usually at least 640x480 pixels) with a passable time resolution (25-30 frames per second).

There are several things to consider when preparing to make frames for videotape. These are:

(1) The frame rate of recording media.
(2) The capabilities of the video tape format
(3) The appropriate image size and quality.
(4) The aspect ratio of the images.
(5) Color selection.
(6) Computational capacity.
(7) Storage capacity.

4.1. Frame Rate

The television and video industry has not settled upon a world-wide standard for the encoding of video pictures. Some of the basic elements are common to all the standards.[10] Each frame of video is comprised of two ''fields.'' A field consists of either the even numbered scanlines or the odd numbered scanlines from the frame.[11] To display the frame, first the field made up of the odd numbered scanlines is displayed. When this is completed, the field containing the even numbered scanlines is displayed.

The NTSC video encoding system (used in the United States and Japan) displays approximately 30 frames per second.[12] This means that 60 fields are displayed per second. The flicker that would ordinarily be perceived at 30 frames per second is substantially reduced. The PAL (Western Europe and Australia) and SECAM (France & former Soviet Union variant) encoding formats display 25 frames per second or 50 fields per second.

4.2. Video Tape Formats

It is worth noting that not all videotape formats are created equal. Some are capable of retaining more image detail than others. The table below lists the number of side-by-side alternating black-and-white vertical lines which can be discerned in 3/4 the width of a single frame of video using each of the video recording formats. This is referred to as ''video lines of resolution'' or ''TV Lines'' in the video technology literature.

Resolution of Video Recording Formats	
Format	VLines
VHS (1/2 inch)	240
3/4 inch	260
3/4 inch SP	380
SVHS (1/2 inch)	400
Hi8	400
BetaCam SP	400
D2 Digital Video	440

[10] If one ignores the evolving standards for High Definition Television (HDTV),

[11] For purposes of simplicity and practicality many details of the true structure of video imaging will be blatantly glossed over. Such details are beyond the scope of this paper. Purists are asked to remain quiet. Novices are directed to **[Kennedy91]** and **[Benson85]**.

[12] Color NTSC actually uses 29.97 frames per second. The difference is only important when creating precisely timed sequences.

There are other differences between the various formats listed. The table is roughly ordered from lowest to highest image quality. If possible, avoid using formats closer to the top of the table for making original recordings. Choose the best format available to you for making original recordings. It is usually easy to duplicate an original to a less capable format for distribution. Plan on making all duplicates from original recordings if possible. Second and third generation copies made with the analog recording formats will show degradation in color (bleeding and noise), image stability (straight lines become wavy), and resolution. If the distribution copies of your animation must be provided on VHS, keep in mind that fine details in the scene may be lost when the transfer is made to this format.

4.3. Image Size and Quality

In order to know what size images need to be computed it is necessary to know something about the hardware that will be used to convert the images to a video signal. The video encoding system found in the Silicon Graphics Iris (tm) family of workstations encodes a 640x480 pixel image. The Abekas A60 Digital Video Disk system encodes an image that is 720x486 pixels in size. Other video encoding hardware may use other resolutions. You should determine the image size encoded by your hardware.[13]

Frames should actually be computed at twice the resolution that will be used for the encoding (e.g. 1280x960 instead of 640x480) The ''pixhalve'' program is used to reduce frames to the appropriate size for video encoding. The filter kernel used by pixhalve reduces sampling artifacts such as the pixel staircase effect on diagonal lines (a.k.a. ''jaggies'') in the final image. It also attempts to ''spread'' single pixel details slightly (such as specular glints of light off surfaces). This slight spreading helps to compensate for the fact that current video encoding and recording techniques have difficulty with such fine details.

The ''-J1'' option should also be given to rt. This turns on the ''ray jittering'' feature of rt. Ordinarily, rt traces rays through the center of each pixel. This can lead to visual artifacts resulting from the regular sampling grid. When ray jittering is enabled, rt picks a different location inside each pixel at which it traces the ray. This reduces the regularity of the sampling grid, and hence any artifacts that might result.

4.4. Image Aspect Ratio

The rt program fits a viewing cube with 1:1 aspect ratio in model coordinates to the dimensions of the image being created. This means that whenever a non-square image is created (such as a 4:3 aspect ratio picture for use in video recording) the image is distorted. This is most noticeable when computing images of involving circular objects such the sphere, rcc (right circular cylinder) or the torus. This becomes a problem when preparing images for visual analysis and animation. The ''-V'' option to rt provides the solution to the problem. The following shell script demonstrates the use of the this option.

```
#!/bin/sh
#   compute 640x512 image to be displayed on system with square
#   pixels. compensate for the distortion of the viewing cube.
rt -M -w640 -n512 -V640:512 -o img.pix moss.g 'all.g' 2>> img.log <<EOF
viewsize 1.421157820291206e+02;
eye_pt 20.21 -71.12 26.12;
orientation .707 0.0 0.0 .707;
start 0;
end;
EOF
```

This option should be used whenever non square images are being prepared, or when the images will be displayed on a system with pixels which do not have a 1:1 aspect ratio. The option allows the user to compensate for the distortion introduced in these situations. Note that in the example above, ''-V5:4'' would have produced the same results. Many users find that using the dimensions of the image being computed is easier than computing the aspect ratio in small integers.

[13] While there is no ''one size fits all'' image size, the 640x480 resolution is common.

4.5. Color Selection

Video has a very limited capacity for color. It also does not have equal capacity for storing the primary colors red, green, and blue. This usually comes as a great disappointment to people accustomed to looking at computer images in rich detail on workstation monitors. Whenever possible, the colors for objects in a video scene should be chosen to match the abilities of the video system being used.

The first rule of thumb is: if an image or scene is going to be recorded on video, the colors should be chosen by viewing the scene through the video encoder, not on the workstation monitor. A good approach is to render several images from your animation sequence, and record long runs of them in the same manner that will be used for the final animation frames. View this test recording to see how colors will appear in the final result.

Blue objects in a scene are most likely to have visual noise to their appearance. A large smooth blue surface is likely to look ''grainy'' in the final result. Small blue details will get lost in the noise.

Red objects are most likely to contribute to smearing or color-bleeding in the final result. Where possible, avoid placing bright or predominantly red objects next to areas of important detail.

One other consideration worth mentioning is that no color in the scene should have more than a 75 percent level of saturation. While this is rarely a problem with images generated by rt, it is a frequent ''beginner's mistake'' when preparing title frames or other hand-painted images. The ''fbcolor'' program makes the viewing and selection of specific colors easy. It also reports the saturation level of the colors it displays as a number from 0 to 255. You should avoid using any color which fbcolor reports as having a saturation over 191. If you should notice that an object image created by rt lacks depth or surface detail (such as a lighted cylinder which shows no shading along the curved surface), check to see what color it was given in the model. Reducing the saturation of the color given the object should help improve its appearance.[14]

4.6. Computational Requirements

There is a tradeoff to be made between the number of frames computed (and hence the computational requirements) for a sequence of a given duration and the quality of the motion in the result. Larger numbers of frames (and more CPU time expended in their creation) result in smoother motion. Fewer frames per second result in more ''jerky'' movement of the camera and objects.

The smoothest motion is obtained when an image is computed for each field of the final animation. Under NTSC this produces the appearance of 60 separate images per second. This is just above the threshold at which most individuals perceive flicker.

The scanlines for each field must be extracted from the images. The two fields which form a video frame are combined together to create the video frames using the pixfields program. These are then encoded to video and recorded. Besides the computational requirements, the only drawback is that sequences created in this manner do not look as pleasing when used with ''freeze frame'' features common on videotape machines.

The most common compromise is to compute one image for each video frame of the final result. There is no requirement for field compositing, and the image obtained from a ''freeze frame'' of the videotape machine is pleasing. Under NTSC this produces the effect of 30 frames per second.

If each frame is shown for two or more successive frame times, the perception of motion begins to suffer. For any finished product it is advisable to use each image for three or fewer frame-times.

It can take hours, days, or weeks of time for rt to create all of the images in an animation sequence. As a result, there is a greater probability that rt will be interrupted before it can finish the task. If 200 images of a 300 image animation were completed before rt was interrupted, it would be a waste to re-create those images when rt is restarted. Therefore rt was given some heuristics to allow it to avoid duplicating results and wasting CPU time.

Whenever the rt program is asked to create an image, it first looks to see if the image already exists. If the image file exists and is read-only, the image is assumed to be completed and rt moves on to the next image. If the image file exists and is writable, then rt looks for black pixels (R,G,B color 0,0,0) in the image. Since rt never creates image pixels with this exact color, any such pixels it finds in the image are recomputed. If the file is smaller

[14] You should also make certain that the image is viewed with the correct gamma compensation for the display device. See the manual page for fbgamma for more information.

than the final image should be, rt computes only the missing portion to complete the file.

The amount of time required to create the frames of a particular animation sequence can be reduced by employing more than one machine to perform the task. This can be done by giving multiple machines portions of the sequence to create. If the sequence is divided up into frames, each machine can work on a set of frames. Alternatively, the program ''remrt'' can be used in place of the ''rt'' program. Remrt distributes the computational load for each frame across a large number of machines. Each machine is assigned a portion of the image to prepare. The result is assembled on a single machine. For a more complete discussion of remrt see **[Muuss90a]**.

4.7. Storage Requirements

Animation frames can require a great deal of disk space. If the frames for a 20 second animation at 30 frames per second were computed at a resolution of 1440x972 pixels per image, the result would be approximately 2.4 gigabytes of storage. Fortunately, it is possible to reduce this requirement substantially. The rt program can be given a shell command to execute from within the animation script. For example, the file ''ell.proto'' could have looked like this:

```
viewsize @0;
eye_pt @1 @2 @3;
orientation @4 @5 @6 @7;
start @(line);
clean;
anim all.g/ellipse.r matrix rmul
        1 0 0 @8
        0 1 0 @9
        0 0 1 @10
        0 0 0 1;
end;

! framedone.sh ell.pix @(line);
```

After each image is completed, rt would execute the shell command ''framedone.sh'' with the image number for an argument. This shell script can be used to run pixhalve and compress the results and perform other functions. For example:

```
% cat framedone.sh
#!/bin/sh
# Script to be run by rt whenever a frame of an animation is completed.
# Should be invoked from rt animation script as follows:
#
# viewsize 200;
# eye_pt 20. 0.0 83.25;
# orientation 0.0 0.0 .9238 .3826;
# start 3;
# end;
# ! framedone.sh file.pix 3;
#
if [ $# -ne "2" ] ; then
        echo "Usage: $0 basename.pix frame_number" ; exit
fi

echo $0 $*

if [ "$2" -eq "0" ] ; then
        FILE=$1
else
        FILE=$1.$2
fi
```

```
pixhalve -w1440 $FILE > $FILE.sm
chmod -w $FILE.sm
compress $FILE.sm
chmod +w $FILE
mv $FILE.sm.Z $FILE
```

The second "if" statment in this script is necessary because rt does not append a frame number to the filename of the first image in an animation sequence. This preserves compatibility with scripts intended for creating a single frame.

5. Computing Video Frames

With the animation tables used in the moving ellipsoid example from Section 3, we can create all the frames for a full-speed videotape recording. It is important to avoid file name conflicts between the images created for the postage stamp animation and the frames for the video. Either tell rt to create frames with different names, or the postage stamp frames must be renamed or removed before computing the video frames. Failure to do so will cause rt to believe that the postage stamp frames are completed frames for the video animation. In the following example, the frames will be created with the name "ell_vid.pix" to avoid conflict.

The arguments to the "times" command of tabinterp is slightly different from the previous example. The difference is the number of frames per integer time step (seconds) is increased from 3 to 30.

```
% tabinterp << EOF > chans.all
file chans.vsize 0;
file chans.eyept 1 2 3;
file chans.orient 4 5 6 7;
file chans.ellanim 8 9 10;
times 0 8 30;
interp spline 0 1 2 3 4 5 6 7;
interp linear 8 9 10;
EOF
cmd: file chans.vsize 0
chan 0:  File 'chans.vsize', Column 1
cmd: file chans.eyept 1 2 3
chan 1:  File 'chans.eyept', Column 1
chan 2:  File 'chans.eyept', Column 2
chan 3:  File 'chans.eyept', Column 3
cmd: file chans.orient 4 5 6 7
chan 4:  File 'chans.orient', Column 1
chan 5:  File 'chans.orient', Column 2
chan 6:  File 'chans.orient', Column 3
chan 7:  File 'chans.orient', Column 4
cmd: file chans.ellanim 8 9 10
chan 8:  File 'chans.ellanim', Column 1
chan 9:  File 'chans.ellanim', Column 2
chan 10:  File 'chans.ellanim', Column 3
cmd: times 0 8 30
cmd: interp spline 0 1 2 3 4 5 6 7
cmd: interp linear 8 9 10
performing interpolations
writing output
```

The new prototype contains an invocation of the "framedone.sh" script to process each image as it is completed.

```
% cat ell.proto.2
viewsize @0;
eye_pt @1 @2 @3;
orientation @4 @5 @6 @7;
start @(line);
clean;
anim all.g/ellipse.r matrix rmul
        1 0 0 @8
        0 1 0 @9
        0 0 1 @10
        0 0 0 1;
end;
! framedone.sh ell_vid.pix @(line);

% tabsub ell.proto.2 chans.all > ell.rtanim
```

It is likely to be quite a while before all the images are computed. The rendering should be done as a batch job or detached process using a script similar to the ''ell2.rt'' script below.

```
% cat ell2.rt
#!/bin/sh
rt -M -w1440 -n 972 -V1440:972 -J1 -o ell_vid.pix moss.g all.g 2>> ell.log < ell.rtanim

% ell2.rt &
```

For instance it took almost ten hours to compute and process the 241 1440x972 images of our trivial geometry for an 8 second animation on a Silicon Graphics 4D/280. The final compressed images occupied approximately 39 MB of disk space.

6. Recording Videotape

Once all the frames have been computed it is time to record them onto videotape. There are a variety of tools and techniques for accomplishing this task. See **[Kennedy91]** for a description of some of the variety of equipment which has been used at the Army Research Laboratory for this purpose. Only one of these techniques will be covered here.

The Abekas A60 digital video disk stores 25 seconds or 750 frames of video on a high-performance disk. Frames on this disk can be played at full video speed. The A60 provides video output as both CCIR 601 digital video and an analog signal that is either R,G,B or Y,R-Y,B-Y. This analog signal can be readily converted to a format for input to a video tape recorder.

Frames can be stored on disk either from a video input or by loading individual frames through an ethernet interface. It is the latter which makes the device particularly useful in creating computer-generated video productions.

Under BRL-CAD, access to the Abekas A60 is achieved through the framebuffer library.[15] Conceptually, the A60 has 750 unique framebuffers. The framebuffer device ''/dev/ab'' supports the A60. There are two very important options to this particular framebuffer. The first specifies the host address of the A60 on the TCP/IP network. The host name or address for the A60 is specified by an at-sign ''@'' followed by the appropriate name or address. For example, the framebuffer device ''/dev/ab@vidisk'' specifies that an A60 connected to the network with the host name of ''vidisk'' should be used. The individual frame (or framebuffer) within the A60 is specified by a sharp-sign ''#'' followed by the integer frame number. Thus the full specification of a framebuffer device might be:

```
/dev/ab@vidisk#27
```

This specifies the frame 27 on the Abekas A60 with the host name ''vidisk''. This can be used with any of the framebuffer utilities, such as the ''pix-fb'' utility shown on the next page.

[15] See the manual page for LIBFB supplied with the BRL-CAD distribution or **[Muuss90a]** for more information about the framebuffer support library in general.

```
% pix-fb -F/dev/ab@vidisk#27 -w 720 -n 486 ell_vid.pix.27
```

This loads the image file "ell_vid.pix" into frame number 27 on the A60 called "vidisk".

Two more options worth noting are the "o" and "v" options to the A60 framebuffer device. The "o" option indicates that this is an "output only" access of the framebuffer. The overhead of retrieving the image already in the framebuffer is skipped when this option is specified. The "v" option turns on verbose logging of the access to the framebuffer. This is primarily useful to enable the user to watch the progress of the framebuffer access.

This can be generalized into a shell loop to load the 241 "ell_vid.pix" images into the A60.

```
% mv ell_vid.pix ell_vid.pix.0
% foreach i (`loop 0 240`)
? pix-fb -F/dev/abov@vidisk#$i -w 720 -n 486 ell_vid.pix.$i
? end
```

In this example successive images are loaded into successive video frames (framebuffers) in the A60. As each frame is loaded, the user will see output indicating that the image is converted to YUV format and then loaded into the A60.

Once the entire sequence (or a 25 second segment) has been loaded into the A60, the sequence can be played and captured using a video tape recorder. A collection of 25 second video sequences can be edited together using a television studio or post-production facility to create longer sequences.

7. Conclusion

Visualization of models in BRL-CAD need not be restricted to viewing static images. With the use of tabinterp and tabsub the task of creating motion picture sequences becomes quite manageable. The result is that models can be viewed from a variety of positions as they move, change, and evolve over time.

The tools discussed represent only one conceptual approach to the problem of specifying frames to be generated for an animation sequence. With the advent of ubiquitous desktop graphics displays, it should be possible to create the sequences using a more visual and interactive approach. Creating tools for this remains an area for further work.

The authors wish to thank Mike Muuss and Phil Dykstra for creating the animation capabilities within rt and mged. The authors also thank Mike Muuss for providing the documentation for tabinterp and tabsub that appears in Appendix A and Appendix B respectively.

INTENTIONALLY LEFT BLANK.

8. References

[Benson85]

Television Engineering Handbook, Benson, K. Blair ed. McGraw-Hill Book Company, 1986. ISBN 0-07-004779-0

[Deitz89]

Deitz, P. H., W. H. Mermagen, Jr., and P. R. Stay. "An Integrated Environment for Army Navy and Air Force Target Description Support," Proceedings of the Tenth Annual Symposium on Survivability and Vulnerability of the American Defense Preparedness Association, Naval Ocean Systems Center, San Diego, CA, May 10-12, 1988.
Also in *The Ballistic Research Laboratory CAD Package Release 4.0 Manuals, Volume I, BRL-CAD Philosophy* Page V1S04A00

[Foley92]

Foley, James D., Andries van Dam, Steven K. Feiner, John F. Hughes, *Computer Graphics, Principles and Practice,* Addison-Wesley Publishing Company, 1992. ISBN 0-201-12110-7
Note: Chapter 5, Section 6 presents homogeneous coordinate systems.

[Kennedy91]

Kennedy, Charles M. "Video Hardware for Making Movies," Proceedings of the 1991 BRL-CAD Symposium, Aberdeen Proving Ground, Maryland, May 7-9 1991.
Also in *The Ballistic Research Laboratory CAD Package Release 4.0 Manuals, Volume V, Analyst's Manual* Page V5S14A01

[Muuss88a]

Muuss, M. "Understanding the Preparation and Analysis of Solid Models", *Techniques for Computer Graphics,* Rogers, D. F., and R. A. Earnshaw, ed. Springer Verlag, New York, pages 109-172. ISBN 0-387-96492-4 also ISBN 3-540-96492-4
Also in *The Ballistic Research Laboratory CAD Package Release 4.0 Manuals, Volume V, Analyst's Manual* Page V5S08A05.

[Muuss88b]

Muuss, M. and P. Dykstra. "The RT Lighting Model." Proceedings of the BRL-CAD Symposium '88, Ballistic Research Laboratory, Aberdeen Proving Ground, Maryland, June 28, 1988.
Also in *The Ballistic Research Laboratory CAD Package Release 4.0 Manuals, Volume V, Analyst's Manual* Page V5S10A03.

[Muuss90a]

Muuss, M. J. "Workstations, Networking, Distributed Graphics, and Parallel Processing" *Computer Graphics Techniques: Theory and Practice* Rogers, D. F., and R. A. Earnshaw ed., Springer-Verlag, 1990. ISBN 0-387-97237-4
Also in *The Ballistic Research Laboratory CAD Package Release 4.0 Manuals, Volume V, Analyst's Manual* Page V5S06A02.

[Newman79]

Newman, W. M., and R. F. Sproul. *Principles of Interactive Computer Graphics, 2nd ed.,* McGraw-Hill, New York, 1979. ISBN 0-07-046338-7
Note: Chapters 22 and 23 present matrix transformation and homogeneous coordinates.

[Rogers90]

Rogers, D. F., and J. A. Adams. *Mathematical Elements for Computer Graphics, 2nd ed.*, McGraw-Hill, New York, 1990. ISBN 0-07-053529-9 (hard cover) ISBN 0-07-053530-2 (soft cover)
Note: Chapters 2 and 3 present matrix transformation and homogeneous coordinates.

[Shoemake85]

Ken Shoemake ''Animating Rotation with Quaternion Curves'' *Computer Graphics,* Vol. 18 No. 3, July 1985, SIGGraph '85 Proceedings

Appendix A:
Tabinterp

INTENTIONALLY LEFT BLANK.

NAME

tabinterp – combine and interpolate multiple data files to create an animation script

SYNOPSIS

tabinterp >table.final

DESCRIPTION

tabinterp reads a series of commands from standard input which designate what parts of various data files should be used as input tables for various *channels* of animation parameters. Commands may extend across multiple lines, and are semi-colon (';') terminated. Each channel is then interpolated using one of a variety of interpolation techniques to provide an output table which has one line for each time step.

The overall notion is based on parameter *tables*. Each table is arranged so that every row (line) represents the state of some set of parameters at a given *time*. Each column of the table represents a single parameter, or data *channel*, with the left-most column always representing *time*.

The first task in preparing to use is to assign specific purposes to each channel in the output table. For example, channels 0, 1, and 2 might be used to represent the X, Y, and Z positions of an object, respectively, while channels 3, 4, and 5 might be used to represent the "aim point" of the virtual camera, while channel 6 might be used to represent the brightness of one of the objects or light sources, and channel 7 might be used to represent the zoom factor (viewsize) of the virtual camera. Once the channel assignment has been decided upon, the source file containing the table of raw values for each channel must be identified. Several output channels may get their raw values from different columns of a single input table (file). Up to 64 columns of input may appear in an input table.

For each file which contains an input table, the **file** command is given to load the necessary columns of raw values into the output channels. If a channel number in the list is given as a minus ('-'), that input column is skipped. Using the output channel assignments given above as an example, if an input table named "table1" existed which consisted of five columns of values representing (time, brightness, objX, objY, objZ), then these values would be loaded with this command:

```
file filename chan_num(s);
file table1 6 0 1 2;
```

This command indicates that from the file "table1", the current time and four columns of parameters should be read into the raw output table, with the first input column representing the time, the second input column representing the value for output channel 6 (brightness), the third input column representing the value for output channel 0 (objX), etc. Each row of the input file must fit on a single (newline terminated) line of text, with columns separated by one or more spaces and tabs.

After all the **file** commands have been given, it is necessary to define over what range of time values found in the raw output table will be processed, and how many rows of interpolated output should be produced for each second (time unit) in the input file. This can be thought of as the "frames per second" rate of the interpolation, and is usually set to 24 for film (cine) work, 30 for NTSC video, and 60 for field-at-a-time NTSC video. Any positive integer value is acceptable. (In fact, any time unit can be used, as the time channel is dimensionless. Nothing depends on the units being seconds.) For example, the command:

```
times start stop fps;
times 1 7.3 24;
```

would cause *tabinterp* to process data values from time 1 second to 7.3 seconds, producing 24 output rows uniformly separated in time for the passage of each second.

After the **times** command has been given, it is necessary to associate an interpolator procedure or a "value generator" procedure with each output channel. The available interpolator procedures are: **step, linear, spline, cspline,** and **quat**. For example, the command:

```
interp type chan_num(s);
interp linear 3 4 5;
```

would indicate that output channels 3, 4, and 5 (representing the camera aim point) would be processed using linear interpolation. If only a starting and ending values are given in the input (*i.e.* the input file had only two rows), then this is an easy way of moving something from one place to another. In this case, if more than two input rows had been provided, there would be a noticeable "jerk" as the camera passed through each of the input parameter values, an effect which is rarely desired. To avoid this, the **spline** interpolator can be used, which fits an interpolating spline (with open end conditions) through the given data values, resulting in smooth motion. If the starting and ending values are the same, a continuous spline (with closed end conditions) can be used instead by specifying **cspline.** Both of the spline interpolators require at least three rows to have been provided in the input file.

If the output values are to "jump" from one input value to the next, (*i.e.* no interpolation at all is desired), then specify **step.** This can be useful for having lights switch between several intensities (for example, a 3-way bulb with 30, 70, and 100 watt settings), or for having objects "teleport" into position at just the right moment.

The interpolation method indicated on the **interp** command is assigned to all the output channels listed. One exception to this rule is the **quat** (Quaternion) interpolator. Quaternions are used to describe an orientation in space, and can be most easily thought of as containing a vector in space, from which they obtain a pointing direction, and a "twist" angle around that vector. To do this, quaternions are processed in blocks of four channels, which must be numbered sequentially (*e.g.* channels 7, 8, 9, 10). Giving the command

```
interp quat 7 15;
```

assigns the quaternion interpolator to two blocks of four channels, the block starting with channel 7 (*e.g.* channels 7, 8, 9, 10), and the block starting with channel 15.

tabinterp is strictly an interpolator. It will not extrapolate values before the first input value, nor after the last output value. The first or last value is simple repeated.

In addition to interpolation, it is possible to specify rate and acceleration based output channels. In cases where the exact running time of a scene is not known, the **rate** and **accel** commands can be quite useful. One command is given for each output channel. For example,

```
rate chan_num init_value incr_per_sec;
rate 6 1.5 0.5;
```

says to make channel 6 a rate based channel, with the initial value (at time=0) of 1.5, linearly increasing with an increment of 0.5 for the passing of every additional second. In this case, the value would be 2.0 at time=1, 2.5 at time=2, and so on. This can be used to establish linear changes where it is the increment and not the final value that is important. For example, the rotation angle of a helicopter rotor could be specified in this way.

Similarly, the command

```
accel chan_num init_value mult_per_sec;
accel 5 10 2;
```

says to make channel 5 an acceleration based channel, with the initial value at time=0 of 10.0, which is multipled by 2 for every additional second. In this case, the value would be 20.0 at time=1, and 40.0 at time=2. This can be useful to create constant acceleration, such as a car accelerating smoothly away from it's position at rest in front of a stop sign. If the initial value is zero, all subsequent values will also be zero.

Sometimes it is desirable to create an output channel which looks ahead (or behind) in time. For example, a good way to animate a rocket flying on a complex course would be to simply animate the position of the base of the rocket, and then look ahead in time to see where the rocket is going to go next in order to determine where to aim the nose of the rocket (by rotating it). This kind of lookahead is easily implemented using the **next** command. (See also the **fromto** directive in which is used in conjunction with this). The command

```
next dest_chan src_chan nsamp;
next 4 5 +3;
```

says to fill channel 4 with the values that will be present in channel 5 at 3 output rows later on. Negative values are also permitted. Since the lookahead is defined in terms of output rows rather than time steps, this means that the values generated for this column will change as the frames per second (fps) value on the **times** command is changed. This is almost always the effect which is desired, since as the temporal resolution of the interpolation is increased, the accuracy of the look-ahead will increase as well. However, if the effect desired is one of "have the camera track where the main actor was three seconds ago", then the number of steps given here will have to be changed when the fps value is changed. Be careful of the values generated for the last (or first) *nsamp* output rows. Looking forward or backward in time beyond the bounds of the interpolation will retrieve the last (or first) output values. So it takes *nsamp* output rows to "prime the pumps".

Whenever a pound sign ('#') is encountered in the command input, all characters from there to the end of the input line are discarded. This is the same commenting convention used in the Bourne shell,

When *tabinterp* encounters an end-of-file on it's standard input, it computes the requested interpolations, and writes the output table on standard output. If no values have been assigned to an output channel, then the value given is a single dot ('.'). This preserves the positional white-space-separated columns nature of the output table. If this column is read as a numeric value by a downstream program, it will be accepted as a valid floating-point zero.

As an aid to debugging, it is possible to dump the raw values of columns of the output table before the interpolation is run:

```
idump;
idump chan_num(s);
```

If no output channel numbers are given, all channels are dumped, otherwise only the indicated channels are dumped.

The **help** command can be given to get a list of all available commands. (Don't forget the semi-colon).

EXAMPLE

What follows here is a Bourne shell script which will generate two input tables using "here documents", and will then produce an interpolated output table of 8 channels.

```
#!/bin/sh
cat << EOF > table.aim
-1  0 0 0   42 250
3   1 2 3   28 300
7   3 4 5   17 350
EOF
cat << EOF > table.obj
0   17 38 44
2   43 47 3
4   99 23 18
EOF
tabinterp << EOF > table.final
# Channel allocations:
#       0,1,2objX, objY, objZmain actor position
#       3,4,5aimX, aimY, aimZcamera aim point
#       6light brightness
#       7viewsize
#
# Input table column allocations:  time, aimX, aimY, aimZ, junk, viewsize
file table.aim 3 4 5 - 7;
```

```
#
# Input table column allocations:  time, objX, objY, obxZ
file table.obj 0 1 2;
# Channel 6 is not read in here, but is rate base.
#
# Tstart, Tstop, fps
times 0 4 30;
#
# Assign interpolators to output channels
rate 6 1000 50;# 1000 lumen bulb keeps getting brighter...
interp linear 0 1 2;
interp spline 3 4 5;
interp spline 7;
EOF
```

Try clipping this example out of the manual page (usually found in /usr/brlcad/man/man1/tabinterp.1) and running it. This example will be continued in the manual page for

POST PROCESSING

Because both the input and output tables consist of a single line of text for each time step, many of the standard UNIX tools can be brought to bear to assist in creating an animation. To visualize the exact position taken by the aim point in the example (output channels 3, 4, 5), a UNIX-plot file of that trajectory can be created with:

```
cut -f5,6,7 table.final | xyz-pl > aim.pl
cut -f1,5,6,7 table.final | txyz-pl > aim.pl
```

Similarly, the position of the main object can be viewed with

```
cut -f2,3,4 table.final | xyz-pl > obj.pl
```

tabinterp uses 0-based column numbering, while *cut* uses 1-based column numbering. Also, the first output column from *tabinterp* is always the time. The 0-th data column comes second.

The plot file just created can be viewed using or or it can be viewed in by giving the command

```
overlay aim.pl
```

to *mged.* If the model geometry is brought into view using the *mged* **e** command, then the camera aim track (or any other spatial parameter) can be viewed in direct relationship to the three dimensional geometry which is going to be animated.

PREPARING INPUT TABLES

The **savekey** and **saveview** commands can be very useful for creating the input tables necessary for driving The details of doing this are beyond the scope of this manual page.

The command can also be useful for routing through the output files of existing scientific analysis programs, and extracting the few gems of data burried in the heaps of "printout".

SEE ALSO

tabsub(1), xyz-pl(1), txyz-pl(1), cut(1), paste(1), rt(1), mged(1)

DIAGNOSTICS

In it's present form, the program is a bit verbose, reporting on the progress of each command on standard error. This behavior will probably be placed under control of a **–v** flag in a future version.

BUGS

You can't *grep* dead trees.

AUTHOR

Michael John Muuss

SOURCE

The U. S. Army Research Laboratory
Aberdeen Proving Ground, Maryland 21005

BUG REPORTS

Reports of bugs or problems should be submitted via electronic mail to <CAD@BRL.MIL>.

INTENTIONALLY LEFT BLANK.

Appendix B:
Tabsub

INTENTIONALLY LEFT BLANK.

NAME

tabsub – macro expand an input table into an animation script

SYNOPSIS

tabsub template_file <table.final >>script

DESCRIPTION

tabsub takes as input a data table on standard input (such as might have been produced by or similar tool), and a template file named on the command line. For each row (line) of the input table, one complete copy of the template file is output on standard output. As the template is output, any macro invocations in the template file are replaced with the data values from the input table's current row. In the input table, any blank lines or lines with a pound sign ('#') as the first character are ignored, allowing comments to be added to the input table.

Macro invocations in the template file all begin with an at-sign ('@'). In order to send an at-sign through to the output, a second at-sign must immediately follow it, *e.g.* when '@@' is encountered in the template, a single '@' is output. To output the data value found in a given channel in the current input row of the data table, the at-sign is followed by the channel number, *e.g.* to output the value in channel four, specify '@4', and to output the value in channel 42, specify '@42'. In some circumstances it my be desirable to highlight the difference between channel value substitution, and literal numeric values. To facilitate this, the channel number may be enclosed in parenthesis to explicitly delimit the macro invocation. For example, channel four could also be specified as '@(4)', and channel 42 as '@(42)'. This second notation is generally preferred.

The *tabsub* program is intended primarily for creating scripts relating to animation. To facilitate this, a variety of more complex macros also exist.

```
@(line)
```

will output the row (line) number of the input table which is currently being processed, with the first line being numbered zero. This is useful for creating frame numbers, or other sequence tags in the output.

```
@(time)
```

will output the time value which is always found in the left-most column of the current row.

The more complex macros can also take arguments. If the first character of an argument is an at-sign ('@') (or percent-sign ('%'), for backwards compatibility), then the number that follows signifies an input channel substitution as before. Otherwise the value is taken literally.

The **rot** macro is used to convert three Euler angles given in degrees into a rotation expressed as a 4x4 homogeneous transformation matrix.

```
@(rot x_angle y_angle z_angle)
```

The arguments may be either numeric constants, column value macros, or a combination of both. The matrix is generated by calling the routine **mat_angles** which performs the rotation around the Z axis first, then Y, then X. For example, the macro

```
@(rot 0 0 45)
```

creates the following matrix, a 45 degree rotation about Z:

```
7.071067812e-01 -7.071067812e-01 0.000000000e+00 0.000000000e+00
7.071067812e-01 7.071067812e-01 -0.000000000e+00 0.000000000e+00
0.000000000e+00 0.000000000e+00 1.000000000e+00 0.000000000e+00
0.000000000e+00 0.000000000e+00 0.000000000e+00 1.000000000e+00
```

Similarly, the macro

```
@(rot @4 @5 90)
```

creates a rotation matrix where the angle of rotation around X is taken from input channel four, the Y angle is taken from input channel five, and the Z angle is fixed at 90 degrees.

The **xlate** macro converts three distances (which must be specified in millimeters if the output script is destined for processing by or into a translation expressed as a 4x4 homogeneous transformation matrix.

```
@(xlate dx dy dz)
```

The matrix is generated by invoking the C macro **MAT_DELTAS** found in h/vmath.h. For example, the macro

@(xlate 100 -20 300) creates the following matrix:

```
1.000000000e+00 0.000000000e+00 0.000000000e+00 1.000000000e+02
0.000000000e+00 1.000000000e+00 0.000000000e+00 -2.000000000e+01
0.000000000e+00 0.000000000e+00 1.000000000e+00 3.000000000e+02
0.000000000e+00 0.000000000e+00 0.000000000e+00 1.000000000e+00
```

Similarly, the macro

```
@(xlate 13 @7 0)
```

creates a matrix where the origin is translated 13 units (mm) in X, and the number of units found in input channel 7 in Y. No translation occurs in Z.

The **orient** macro combines the operation of the **rot** zand **xlate** macros, and also offers optional scaling. The invocation is one of:

```
@(orient tx ty tz rx ry rz)
@(orient tx ty tz rx ry rz scale)
```

where all rotation is done first, then the translation, and then the scaling (if given).

The **ae** command converts style azimuth and elevation angle given in degrees into a rotation expressed as a 4x4 homogeneous transformation matrix.

```
@(ae azimuth elevation)
```

The matrix is generated by calling the routine **mat_ae**

The **quat** command converts a quaternion into a 4x4 homogeneous transformation matrix.

```
@(quat x y z w)
```

The **fromto** command is used to rotate the given axis to point in the same direction as the vector formed by subtracting the 'next' point from the 'cur' point.

```
@(fromto axis cur_x cur_y cur_z next_x next_y next_z)
```

The *axis* argument must be one of these six strings: *+X, -X, +Y, -Y, +Z, -Z,* where the axis letter is capitalized. The matrix is generated by calling the routine **mat_fromto** where the 'from' argument is derived from the *axis* given, and the 'to' argument is the unit-length difference 'next'-'cur'.

EXAMPLE1

Based upon the example started in the manual page for here is a Bourne shell script which will generate the necessary template file using a "here document", and then process the 8-channel output table left in the file "table.final".

```
#!/bin/sh
# This template will be instantiated once for each frame to be made.
cat << EOF > template

start @(line);
clean;
```

```
lookat_pt @(3) @(4) @(5);
viewsize @(7);
anim all.g/actor.g matrix rmul
 @(xlate @0 @1 @2);
anim all.g/light.r material rparam
 inten=@(6) angle=70 invisible=1;
end;
! framedone.sh actor.pix.@(line);

EOF
# This is the start of the animation script, which will be appended to below.
cat << EOF > script
viewsize 3000;
eye_pt -4.429280979044739e+03 -1.633722950749571e+03 -1.624787858562220e+03;
orientation 5.435778713738288e-01 4.980973490458696e-01 4.564221286261679e-01 4.980973490458693e-01;
#frame data follows
EOF
# Append the data for each frame
tabsub ./template < table.final >> script
```

The frame number is taken from the input table line number, and substituted into the *start* command. The main actor position is taken from channels 0,1,2 and applied (as an "articulation") to the matrix located along the arc between "all.g" and "actor.g" in the *mged* database. The camera (eye) position stays fixed for this animation, but the camera orientation is changed by substituting channels 3,4,5 into the *lookat_pt* command, and the viewsize (zoom lens setting) is changed by substituting channel 7 into the *viewsize* command. The argument to the light region's material property string is replaced with a new string that spells out the current light parameters. After the *end* command, a shell escape is constructed, which will run a script called "framedone.sh" with the given argument (which has been arranged to be the file name of the file that just wrote, so that it can be post-processed, compressed, sent to a video recorder, etc.

Try clipping this example out of the manual page (usually found in /usr/brlcad/man/man1/tabsub.1) and running it.

EXAMPLE2

In the manual page, mention was made of animating the flight of a rocket. This partial example outlines how that might be accomplished.

```
tabinterp << EOF > rocket.final
# Channel allocations:
#   0,1,2position of base of rocket
#   3,4,5next position of base of rocket
#
# Input table column allocations:  time, X, Y, Z
file rocket.table 0 1 2;
#
times 0 4 60;
#
# Assign interpolators to output channels
interp spline 0 1 2;
#
# Get +1 "look ahead" on values, for auto-guidance
next 3 0 1;
next 4 1 1;
next 5 2 1;
EOF
```

```
cat << EOF > rocket.template

start @(line);
clean;
anim all.g/rot.g matrix rmul
 @(xlate @0 @1 @2);
anim rot.g/rocket.g matrix rmul
  @(fromto +Z @0 @1 @2 @3 @4 @5);
end;
EOF
tabsub ./rocket.template < rocket.final >> script
```

The items worthy of note are the use of the **next** command to place the position look-ahead into channels 3,4,5 and the matching use of the *tabsub* **fromto** macro to convert the current and next positions into an appropriate rotation. In this case, the central axis of the rocket as found in the database rises up the +Z axis. Translating the rocket into position is handled one matrix higher up the tree, using the **xlate** macro.

POST PROCESSING

rt style animation scripts can be processed by and by giving the **–M** option on the command line, and providing the script on standard input. For example, the rocket animation might be run like this:

```
rt -M -V4:3 -w1440 -n972 -p90 -o rocket.pix rocket.g all.g < script
```

to produce images in NTSC ("Academy" 4:3) aspect ratio at double the normal resolution, suitable for later processing by

The same animation can be previewed in near real-time using For this example, would be started with

```
  mged rocket.g
```

followed by attaching to an appropriate display device. Then, these commands would be given:

```
  e all.g
  preview script
```

will process each frame as fast as it can, and update the screen.

SEE ALSO

tabinterp(1), xyz-pl(1), txyz-pl(1), cut(1), paste(1), rt(1), mged(1)

BUGS

There is presently a compiled-in limit of 1023 channels in the input table.

AUTHOR

Michael John Muuss

SOURCE

The U. S. Army Research Laboratory
Aberdeen Proving Ground, Maryland 21005

BUG REPORTS

Reports of bugs or problems should be submitted via electronic mail to <CAD@BRL.MIL>.

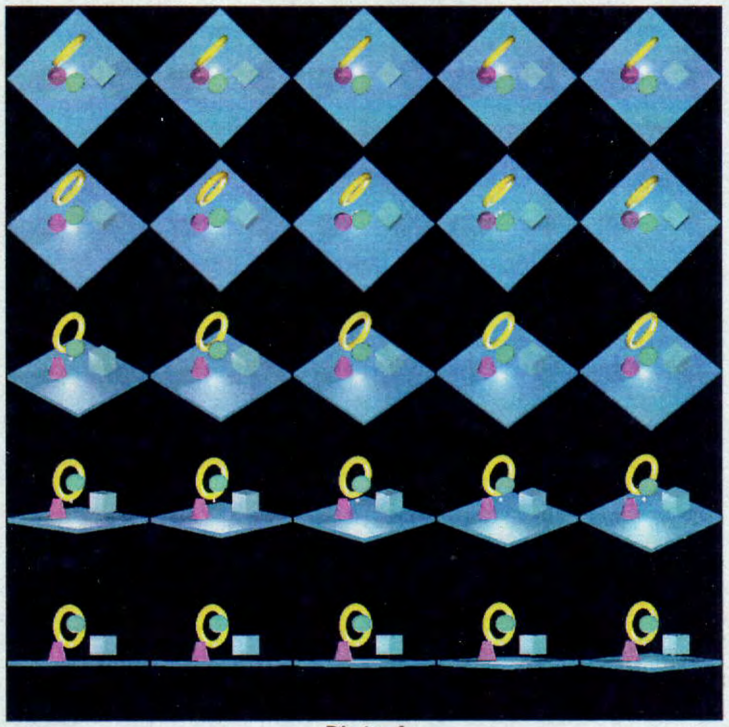
Plate 1

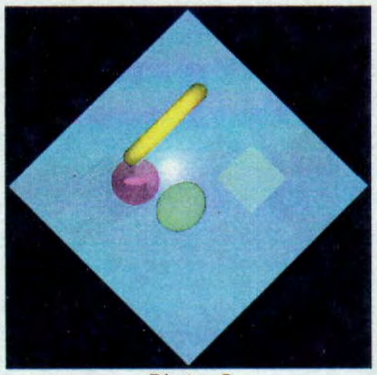
Plate 2

Plate 3

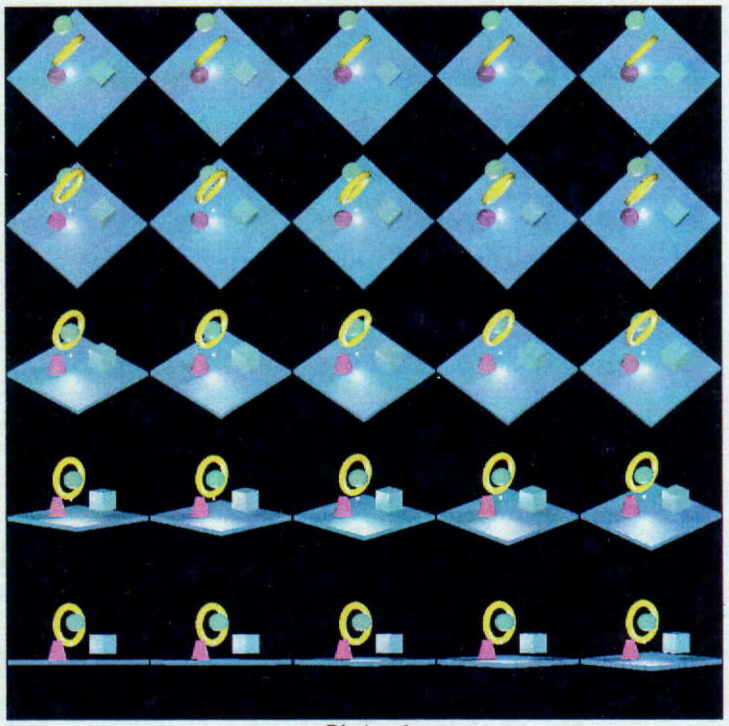
Plate 4

Following page intentionally left blank

No. of Copies	Organization
2	Administrator Defense Technical Info Center ATTN: DTIC-DDA Cameron Station Alexandria, VA 22304-6145
1	Commander U.S. Army Materiel Command ATTN: AMCAM 5001 Eisenhower Ave. Alexandria, VA 22333-0001
1	Director U.S. Army Research Laboratory ATTN: AMSRL-OP-CI-AD, Tech Publishing 2800 Powder Mill Rd. Adelphi, MD 20783-1145
1	Director U.S. Army Research Laboratory ATTN: AMSRL-OP-CI-AD, Records Management 2800 Powder Mill Rd. Adelphi, MD 20783-1145
2	Commander U.S. Army Armament Research, Development, and Engineering Center ATTN: SMCAR-IMI-I Picatinny Arsenal, NJ 07806-5000
2	Commander U.S. Army Armament Research, Development, and Engineering Center ATTN: SMCAR-TDC Picatinny Arsenal, NJ 07806-5000
1	Director Benet Weapons Laboratory U.S. Army Armament Research, Development, and Engineering Center ATTN: SMCAR-CCB-TL Watervliet, NY 12189-4050
1	Director U.S. Army Advanced Systems Research and Analysis Office (ATCOM) ATTN: AMSAT-R-NR, M/S 219-1 Ames Research Center Moffett Field, CA 94035-1000

No. of Copies	Organization
1	Commander U.S. Army Missile Command ATTN: AMSMI-RD-CS-R (DOC) Redstone Arsenal, AL 35898-5010
1	Commander U.S. Army Tank-Automotive Command ATTN: AMSTA-JSK (Armor Eng. Br.) Warren, MI 48397-5000
1	Director U.S. Army TRADOC Analysis Command ATTN: ATRC-WSR White Sands Missile Range, NM 88002-5502
(Class. only) 1	Commandant U.S. Army Infantry School ATTN: ATSH-CD (Security Mgr.) Fort Benning, GA 31905-5660
(Unclass. only) 1	Commandant U.S. Army Infantry School ATTN: ATSH-WCB-O Fort Benning, GA 31905-5000
1	WL/MNOI Eglin AFB, FL 32542-5000
	Aberdeen Proving Ground
2	Dir, USAMSAA ATTN: AMXSY-D AMXSY-MP, H. Cohen
1	Cdr, USATECOM ATTN: AMSTE-TC
1	Dir, USAERDEC ATTN: SCBRD-RT
1	Cdr, USACBDCOM ATTN: AMSCB-CII
1	Dir, USARL ATTN: AMSRL-SL-I
5	Dir, USARL ATTN: AMSRL-OP-CI-B (Tech Lib)

No. of Copies	Organization
	Aberdeen Proving Ground
30	Dir, USARL ATTN: AMSRL-SL-BV, Lee A. Butler (328)(25 cp) AMSRL-WT-WE, Christine A. Murdza (5 cp)

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-313 Date of Report December 1993

2. Date Report Received ____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) ____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) ____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. ____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) ____

	Organization
CURRENT ADDRESS	Name
	Street or P.O. Box No.
	City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

	Organization
OLD ADDRESS	Name
	Street or P.O. Box No.
	City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

BUSINESS REPLY MAIL
FIRST CLASS PERMIT No 0001, APG, MD

Postage will be paid by addressee.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

Director
U.S. Army Research Laboratory
ATTN: AMSRL-OP-CI-B (Tech Lib)
Aberdeen Proving Ground, MD 21005-5066